Immutably Answering Why-Not Questions for Equivalent Conjunctive Queries

Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

Immutably Answering Why-Not Questions for Equivalent Conjunctive Queries

Nicole Bidoit Melanie Herschel Katerina Tzompanaki
Université Paris Sud / Inria, 91405 Orsay Cedex, France
{bidoit/herschel/tzompana}@lri.fr

Abstract

Answering Why-Not questions consists in explaining to developers of complex data transformations or manipulations why their data transformation did not produce some specific results, although they expected them to do so. Different types of explanations that serve as Why-Not answers have been proposed in the past and are either based on the available data, the query tree, or both. Solutions (partially) based on the query tree are generally more efficient and easier to interpret by developers than solutions solely based on data. However, algorithms producing such query-based explanations so far may return different results for reordered conjunctive query trees, and even worse, these results may be incomplete. Clearly, this represents a significant usability problem, as the explanations developers get may be partial and developers have to worry about the query tree representation of their query, losing the advantage of using a declarative query language. As remedy to this problem, we propose the Ted algorithm that produces the same complete query-based explanations for reordered conjunctive query trees.

Categories and Subject Descriptors H.4 [*Information Systems Applications*]: Miscellaneous; D.2 [*Software Engineering*]: Testing and Debugging

Keywords data provenance, query analysis

1. Introduction

The increasing load of data produced nowadays is coupled with an increasing need for complex data transformations that developers design to process these data in every-day tasks, such as data cleaning or data enrichment. These transformations, commonly specified declaratively, may result in unexpected outcomes. For instance, given the query and data of Fig. 1, a developer (or scientist) may wonder why planet Kepler78b is missing from the result, even though he expected or intended it to be part of it. Traditionally, he would repeatedly manually analyze the query to identify a possible reason, fix it, and test it to check whether the missing answer is now present or if other problems need to be fixed.

To help developers during query analysis and debugging, and in particular for answering *why-not questions* as the one used in the above example that asks why some data are *not* part of a result, different algorithms have recently been proposed for relational and SQL queries [2, 3, 6–8] as well as other types of queries (top-k [5], reverse skyline queries [10]). In this paper, we focus on relational queries, for which existing algorithms explain a missing-answer either based on the data (instance-based explanations), the query (query-based explanations), or both (hybrid explanations). Moreover, we focus on solutions producing query-based explanations, as these are generally more efficient while providing sufficient information for query analysis and debugging. Taking a closer look at existing methods, we notice that these return different explanations for reordered query trees. This is due to the fact that these algorithms reason at the level of query tree operators and trace data relevant to the missing-answer, i.e., *compatible data*, through one particular instance of a query tree. On this particular query tree, they identify at which *picky operators* compatible data are lost, and output these.

```
SELECT planet, star
FROM Planets P,
     Stars S
WHERE P.SID = S.SID
AND P.mass <= 1
```

Planet

Planet	Mass	SID
$Earth$	1	1
$Jupiter$	317.8	1
$Kepler37b$	0.01	2
$Kepler78b$	1.7	NULL

Star

SID	Star
1	Sun
2	$Kepler37$

Figure 1. Example query and data

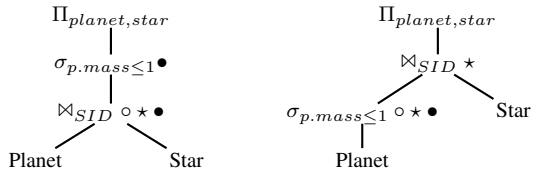

Figure 2. Reordered query trees for the SQL of Fig. 1 and identified algorithms' results (Why-Not ○, NedExplain ⋆, Conseil ●)

Example 1.1. *Consider the SQL query Q and data D of Fig. 1 and assume that a developer wants an explanation for the absence of planet Kepler78b in the query result $Q(D)$, knowing that this planet does not revolve around the Sun. So here, the why-not question is* "Why is tuple ((Planet:Kepler78b, Star:x), $x\neq$Sun) not in $Q(D)$?"*. Fig. 2 shows two possible query trees for Q. It also shows the picky operators that Why-Not [3] (○) and NedExplain [2] (⋆) return as query-based explanations as well as query operators returned as part of hybrid explanations by Conseil [6] (●). It is easy to see that each algorithm returns a different result for each of the two query trees, and in most cases, it is only a partial result as the true explanation of the missing answer is that* both *the selection is too strict for the compatible tuple $(Kepler, 1.7, NULL)$ from table $Planet$ and this tuple does not find any join partner in table Star.*

TAPP '14, June 12–13, 2014, Cologne, Germany.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

To more accurately answer Why-Not questions, we propose the *Ted* algorithm that identifies *all* the *picky operators* of a relational query and explains how they prevent the generation of the desired answer. The proposed explanations take the form of a polynomial, similarly to provenance semi-rings for how-provenance [4] that explain data that exists in a query result. The main asset of this algorithm is that the computed set of query-based explanations (i.e., the *Why-Not answer*) is independent from the query tree representation and is thus not only correct, but also complete w.r.t. the provided definitions. This paper sets the theoretical foundation for computing query-based explanations that are invariant for reordered query trees for conjunctive queries (Sec. 2). We then present *Ted*, a first algorithm computing such explanations and discuss preliminary experiments (Sec. 3). Sec. 4 concludes and discusses future work.

2. Polynomial-Based Why-Not answers

We assume that the reader is familiar with the relational model and tableaux theory [1]. Here, we briefly revisit necessary notions of previously defined Why-Not questions in Sec. 2.1. Sec. 2.2 reviews and extends what has been called compatible data in previous works. Finally, we define the Why-Not answer in Sec. 2.3.

To better illustrate the different aspects of our solution, we resort to a more complex example than the one introduced earlier.

Example 2.1. *Assume a database schema $\mathcal{S}_Q$ consisting of the relations R, S and T and the database instance $\mathcal{I}$ in Fig.3(a). We use a unique annotation Id to identify a tuple of $\mathcal{I}$. Further consider the relational query in Fig. 3(b). The query result includes the tuple $\{R.B{:}5, S.C{:}9, T.D{:}4\}$.*

2.1 The Why-Not Question

Given a query Q over a database schema $\mathcal{S}_Q$[1] and an input instance $\mathcal{I}$, a developer formulates a Why-Not question as a predicate $\mathcal{P}$ that is a disjunction of conditional tuples (c-tuples) [9]. A full definition is available in [2]. Next, we will concentrate on conjunctive queries only and predicates composed of a single c-tuple. The proposed method trivially extends to unions of conjunctive queries and a general predicate $\mathcal{P}$, but we omit a discussion for space constraints.

A c-tuple t_c has the form $(t_v, cond)$, where t_v is a tuple with attribute values being variables and $cond = \bigwedge_{i=1}^{n} pred_i$ is a conjunction of atomic conditions s.t. each $pred_i$ is a comparison between a variable and a constant, or a comparison between two variables. In the following, we will denote the condition associated with a c-tuple t_c as $t_c.cond$ and the set of variables referred to in t_v as $var(t_v)$[2]. Special attention has to be given to the condition associated with the c-tuple t_c. More specifically, we distinguish here between *simple and complex condition*s.

[1] Indeed $\mathcal{S}_Q$ is the query schema of Q as defined in [2], which implies that each relation schema in $\mathcal{S}_Q$ occurs only once in Q.

[2] We also use $var(\cdot)$ to retrieve the set of variables from other structures, e.g., $var(t_c.cond)$ returns the variables for which constraints are specified.

R

A	B	
1	3	Id_1
2	4	Id_2
4	5	Id_3
8	9	Id_4

S

B	C	D	
3	4	5	Id_5
3	8	1	Id_6
5	3	3	Id_7
5	9	4	Id_8

T

C	D	E	
1	4	8	Id_9
3	5	3	Id_{10}
3	3	9	Id_{11}

(a) Sample database instance $\mathcal{I}$

$$\underbrace{\pi_{R.B,S.C,T.D}}_{op_1}(((\underbrace{\sigma_{R.A>3}}_{op_2}(R)) \underbrace{\bowtie_B}_{op_3} (\underbrace{\sigma_{S.C\geq 8}}_{op_4}(S))) \underbrace{\bowtie_D}_{op_5} (\underbrace{\sigma_{T.E\geq 3}}_{op_6}(T)))$$

(b) sample query Q

Figure 3. Sample instance (a) and query (b)

Notation 2.1. *(Simple/Complex condition/c-tuple) An atomic condition $pred_i$ in a condition $cond$ is simple if it compares (a) a variable with a constant or (b) two variables referring to source attributes of the same relation. Otherwise, it is a complex atomic condition. We qualify $cond$ as complex if it includes at least one complex atomic condition, and simple otherwise. Finally, a c-tuple is simple if its condition $cond$ is simple, and complex otherwise.*

Example 2.2. *Given the scenario of Ex. 2.1, we wonder why there is not a result tuple, s.t. the value of $R.B$ is smaller than the one of $T.D$ and on the same time the value of $S.C$ smaller or equal to 9. This Why-Not question is expressed by $t_c{=}((R.B{:}x, T.D{:}y, S.C{:}z), (x{<}y \wedge z{\leq}9))$. In $t_c.cond$, $z{\leq}9$ is a simple condition whereas $x < y$ is a complex condition, because the variables x and y refer to different relations (R and T, respectively). Consequently, t_c is a complex c-tuple.*

2.2 Compatible Data

Intuitively, compatible data designates any source tuples that could have provided data to form the missing answer modelled by t_c. The first step towards answering the Why-Not question consists in identifying these source tuples and more specifically their combinations that form the missing answer in the absence of restrictions in Q. In a second step, discussed in the next section, we will identify query conditions (query operators) that prune these tuple combinations.

Example 2.3. *Continuing Ex. 2.2, $t_c.cond$ implies that the missing-answer is based on a source tuple $t_x{\in}\mathcal{I}_{|R}$, a source tuple $t_y{\in}\mathcal{I}_{|T}$ and a source tuple $t_z{\in}\mathcal{I}_{|S}$ for which $t_x(R.B){<}t_y(T.D)$ and $t_z(S.C){\leq}9$ holds*[3]*. Due to the complex condition, t_x and t_y need to be chosen in correlation with one another, whereas t_z is independent from all others. We obtain (Id_1Id_9), (Id_1Id_{10}) and (Id_2Id_{10}) as compatible tuple concatenation for correlated (t_xt_y), while for t_z each one of the tuples in S, i.e., $Id_5, \ldots, Id_8$ comprises a compatible tuple concatenation.*

Previous approaches [2, 3] consider all compatible tuples independently from each other, e.g., they consider both Id_1 and Id_2 as compatible for t_x. However, Id_2 should lose this property when Id_9 is chosen for t_y, a fact previously ignored. Therefore, in this paper, we introduce the compatibility of a *tuple concatenation* rather than compatibility on isolated tuples. According to our definition, each concatenated compatible tuple (cc-tuple) would have resulted in the missing-answer if it was not pruned by some query operators.

Tableau skeleton. We first define a tableau skeleton $T_{\mathcal{S}_Q}$, which is a set of variable tuples, one for each relation schema in $\mathcal{S}_Q$, such that a variable is not used twice in $T_{\mathcal{S}_Q}$. The relations in $\mathcal{S}_Q$ are also used to identify the rows of $T_{\mathcal{S}_Q}$, as shown in Tab. 1.

	R.A	R.B	T.C	T.D	T.E	S.B	S.C	S.D
R	x_1	x_2						
T			x_3	x_4	x_5			
S						x_6	x_7	x_8

Table 1. Tableau skeleton $T_{\mathcal{S}_Q}$

Mappings. Our subsequent definitions require the mapping functions described and illustrated in Tab. 2. Note that $h_{var(t_c)}$ is used to rename the variables $var(t_c)$ of t_c into variables in $T_{\mathcal{S}_Q}$. Both functions $h_{\mathcal{A}}$ and $h_{var(t_c)}$ are extended to apply on the tableau and the c-tuple conditions respectively. Finally, f naturally extends to concatenated tuples, e.g., $f(Id_1Id_5){=}(R.A{:}1, R.B{:}3, S.B{:}3, S.C{:}4, S.D{:}5)$.

[3] $\mathcal{I}_{|R}$ denotes the instance of relation R and $t(A)$ denotes the attribute value of tuple t on the qualified attribute A.

Function	Purpose	Example
$h_{\mathcal{A}} : \mathcal{A} \rightarrow var(T_{\mathcal{S}_Q})$	Notation for the mapping between attribute names and variables in $T_{\mathcal{S}_Q}$.	$h_{\mathcal{A}}(R.A)=x_1$ $h_{\mathcal{A}}^{-1}(x_1) = R.A$
$h_{var(t_c)}$: $vat(t_c) \rightarrow var(T_{\mathcal{S}_Q})$	Map variables of t_c to variables of $T_{\mathcal{S}_Q}$ associated to the same relation attribute.	$h_{var(t_c)}(x)=x_2$ $h_{var(t_c)}(y) = x_4$ $h_{var(t_c)}(z) = x_7$
$f : ID \rightarrow \mathcal{I}$	Maps a tuple annotation to the actual tuple.	$f(Id_1) =$ $(R.A : 1, R.B : 4)$

Table 2. Mapping functions

Compatible concatenated tuples. We are now ready to define cc-tuples. To this end, we enrich $T_{\mathcal{S}_Q}$ by the condition of $cond$=$h_{var(t_c)}(t_c.cond)$ and a summary S_{t_c}=$h_{var(t_c)}(var(t_c.t_v))$. We thus obtain the *compatibility* tableau T_{t_c}=$(S_{t_c}, T_{\mathcal{S}_Q}, cond)$. For brevity, we will also use the notation T_{t_c}=$(T_{\mathcal{S}_Q}, cond)$ (omitting the summary). A sub-condition $R.cond$ can be associated with row R of T_{t_c} by restricting the conjunction $cond$ to predicates $pred_i$ sharing variables with $var(R)$. So, given $T_{\mathcal{S}_Q}$ in Tab. 1 and the condition $cond$=$(x_2<x_4 \wedge x_7 \leq 9)$, we obtain T_{t_c} in Tab. 3 (ignore the grouping of the rows for now).

		R.A	R.B	T.C	T.D	T.E	S.B	S.C	S.D	cond
$Part_1$	R	x_1	x_2							$x_2 < x_4$
	T			x_3	x_4	x_5				$x_2 < x_4$
$Part_2$	S						x_6	x_7	x_8	$x_7 \leq 9$
	S_{t_c}		x_2		x_4			x_7		

Table 3. Tableau T_{t_c} for our running example

Practically, T_{t_c} models the pattern that a cc-tuple must match. For our example this pattern is: $(R.A{:}x_1, R.B{:}x_2, T.C{:}x_3, T.D{:}x_4, T.E{:}x_5, S.B{:}x_6, S.C{:}x_7, S.D{:}x_8, x_2 < x_4 \wedge x_7 \leq 9)$. This leads to the following definition of a compatible concatenated tuple w.r.t. T_{t_c}.

Definition 2.1. *(Compatible concatenated tuple w.r.t. T_{t_c}) Let $\mathcal{I}$ be an instance of $\mathcal{S}_Q$=$\{R_1, \ldots, R_n\}$ and assume T_{t_c}=$(T_{\mathcal{S}_Q}, cond)$. Let τ=$(Id_1 \ldots Id_n)$ be s.t. $f(Id_i)$∈$\mathcal{I}_{|_{R_i}}$, $\forall i$∈$[1, n]$. Then τ is a compatible concatenated tuple (cc-tuple) w.r.t. T_{t_c} if $f(\tau) \models h_{\mathcal{A}}^{-1}(cond)$. We denote the set of cc-tuples w.r.t. T_{t_c} given $\mathcal{I}$ as $CCT(T_{t_c}, \mathcal{I})$.*

Example 2.4. *For T_{t_c} in Tab. 3 and τ=$(Id_1 Id_5 Id_9)$, it holds that $f(\tau)$=$(R.A{:}1, R.B{:}3, S.B{:}3, S.C{:}4, S.D{:}5, T.C{:}1, T.D{:}4, T.E{:}8)$ and $h_{\mathcal{A}}^{-1}(cond)$=$(R.B<T.D \wedge S.C \leq 9)$. Since $3<4$ and $4 \leq 9$, we get $f(\tau) \models h_{\mathcal{A}}^{-1}(cond)$ and so τ is a cc-tuple w.r.t. T_{t_c}. Totally, we find 12 cc-tuples for our running example.*

2.3 The Why-Not answer

Given the set of cc-tuples $CCT(T_{t_c}, \mathcal{I})$, we define the Why-Not answer using again the tableau skeleton $T_{\mathcal{S}_Q}$, this time to create the tableau T_τ=$(S_\tau, T_{\mathcal{S}_Q}, cond_\tau, cond_Q)$. S_τ=$h_{\mathcal{A}}(f(\tau))$ is the summary while $cond_\tau$ and $cond_Q$ denote rewritten conditions induced by the cc-tuple τ and the query Q, respectively. Due to space limitation we do not provide a formal definition of T_τ. Roughly, $cond_\tau$ embeds τ in the tableau and $cond_Q$ follows from the classical tableau built from Q [1]. We denote $cond_{\tau,R}$ and $cond_{Q,R}$ the restriction of the conditions to the row R.

Example 2.5. *For τ_1=$(Id_1 Id_9 Id_5)$ we obtain T_{τ_1} of Tab. 4.*

Let us now illustrate how T_τ is used to identify *picky* atomic conditions and associated query operators from the query (and thus included in $cond_Q$) that are considered responsible for pruning a cc-tuple τ from the query result.

	R.AR.BT.CT.DT.ES.BS.CS.D	$cond_\tau$	$cond_Q$
R	x_1 x_2	$x_1 = 1 \wedge x_2 = 3$	$x_1 > 3 \wedge x_2 = x_6$
T	x_3 x_4 x_5	$x_3 = 1 \wedge x_4 = 4 \wedge x_5 = 8$	$x_4 = x_8 \wedge x_5 \geq 3$
S	x_6 x_7 x_8	$x_6 = 3 \wedge x_7 = 4 \wedge x_8 = 5$	$x_2 = x_6 \wedge x_4 = x_8 \wedge x_7 \geq 8$
S_τ	x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8		

Table 4. Tableau T_{τ_1}

Example 2.6. *First, focus on τ_1 and the first row R of Tab. 4. The atomic condition x_1=1 in $cond_{\tau,R}$ contradicts the atomic condition x_1>3 of $cond_{Q,R}$. Thus, we say that x_1>3 is a picky condition. The atomic conditions on x_2 in $cond_{\tau,R}$ and $cond_{Q,R}$ are simultaneously satisfied, as x_2=3 $\wedge$ x_6=3 $\wedge$ x_2=x_6 is true.*

In the same way, we identify in the rest of the rows the picky atomic conditions and eventually obtain the set of picky atomic conditions w.r.t. τ_1: $\{x_1>3, x_7 \geq 8, x_4=x_8\}$. Associating these conditions to their respective query operators (see Fig. 3), we obtain the set of picky operators $\{op_2, op_4, op_5\}$.

Notation 2.2. *(Picky operators w.r.t. τ). We define the set of picky conditions w.r.t. τ as PC_τ=$\{c | c \in cond_Q$ and $cond_\tau \not\models c\}$. Each atomic condition c is associated with a query operator op in Q, and we define the set of picky operators w.r.t. τ as PO_τ = $\{op | op$ associated with some $c \in PC_\tau\}$.*

The complete Why-Not answer includes an explanation for the pruning of each cc-tuple τ∈$CCT(T_{t_c}, \mathcal{I})$ and takes the form of a polynomial of query operators.

Definition 2.2. *(Why-Not answer w.r.t. t_c) Given query Q over a database schema $\mathcal{S}_Q$, the instance $\mathcal{I}$ over $\mathcal{S}_Q$, and the compatibility tableau T_{t_c} associated with the Why-Not question t_c, we define the Why-Not answer w.r.t. t_c as* $\sum_{\tau \in CCT(T_{t_c}, \mathcal{I})} \prod_{op \in PO_\tau} op$.

We justify modeling each PO_τ with a product by the fact that in order for τ to 'survive' up to the query result, every single picky operator w.r.t. τ must be 'repaired'. The sum of the products of each $\tau \in CCT(T_{t_c}, \mathcal{I})$ stems from the fact that, if any addend is 'correctly repaired', the associated τ will return the missing answer.

Example 2.7. *In Ex. 2.6 we found that $\{op_2, op_4, op_5\}$ are the picky operators for τ_1, which results in the addend $op_2 * op_4 * op_5$. Applying the same for all 12 cc-tuples in our example, we obtain the final result $op_2 * op_4 * op_5 + 3 * op_2 * op_5 + 3 * op_2 * op_3 * op_4 * op_5 + op_2 * op_3 + 2 * op_2 * op_4 + 2 * op_2 * op_3 * op_5$.*

3. The Ted Algorithm

Alg. 1 presents the *Ted* algorithm that computes the Why-Not answer defined in Sec. 2 for a conjunctive query Q. *Ted* trivially extends to unions of conjunctive queries and a Why-Not question in form of a disjunction of c-tuples (see Sec. 2.1), however, details are omitted due to space constraints.

Ted starts by a preprocessing phase, that consists in creating the tableau skeleton $T_{\mathcal{S}_Q}$ and the tableau T_{t_c} (lines 2 and 3). Then, it determines the set of cc-tuples $CCT(T_{t_c}, \mathcal{I})$ in line 5 before it computes the Why-Not answer (lines 6 – 8). As the computation of the Why-Not answer directly follows from the definitions of Sec. 2.3, we focus our discussion computing $CCT(T_{t_c}, \mathcal{I})$.

To compute the set of all cc-tuples, we could form the cross product of all relations of T_{t_c} (e.g., $R \times T \times S$) and then verify whether each resulting concatenated tuple $(Id_R Id_S Id_T)$ satisfies the condition $t_c.cond$. However, this will result in checking the same conditions numerous times, e.g., the condition $x_7 \leq 9$ will not be checked once for every tuple in relation S, but as many times as there are tuples in the cross product. To improve efficiency, we divide the problem into independent subproblems based on a partitioning of the rows in T_{t_c}.

Algorithm 1: *Ted* algorithm

Input: $\mathcal{S}_Q, Q, \mathcal{I}, t_c$
Output: $Answer$, the polynomial built of the picky operators

Polynomial $Answer = 0$;
Initialize tableau skeleton $T_{\mathcal{S}_Q}$;
Tableau $T_{t_c} \leftarrow createT_{t_c}(T_{\mathcal{S}_Q}, h_{var(t_c)}(t_c))$;
Set $Part \leftarrow$partitioning(T_{t_c});
Set CCT = CompatibleFinder($Part, \mathcal{I}$);
for *(τ: cc-tuple in CCT)* **do**
    $PO_\tau = 1$; initialization of the product of picky operators for τ
    for *(x:variable in $T_{\mathcal{S}_Q}$)* **do**
        $c_\tau \leftarrow$ single atomic condition on x imposed by τ;
        $C_Q \leftarrow$ conditions on x imposed by Q;
        for *(c: atomic condition in C_Q)* **do**
            $c'_\tau \leftarrow true$;
            if *c is a complex condition* **then**
                $x' \leftarrow$ variable compared to x in c_τ;
                $c'_\tau \leftarrow$ single atomic condition on x' imposed by τ;
            if $c_\tau \wedge c'_\tau \wedge c$ **then**
                $PO_\tau \leftarrow PO_\tau * getOperatorForCond(c)$;
        $Answer \leftarrow Answer + Pol_\tau$;
return $Answer$;

Definition 3.1. *(Valid partitioning of T_{t_c}). Assume a partitioning of $T_{\mathcal{S}_Q}$ into k partitions $Part_1, \ldots, Part_k$. This partitioning is* valid *for T_{t_c} if each $Part_i$ is minimal w.r.t. the property: $\forall R \in T_{\mathcal{S}_Q}$, if $R \in Part_i$ and $R' \in T_{\mathcal{S}_Q}$ s.t. $var(R.cond) \cap var(R'.cond) \neq \emptyset$ then $R' \in Part_i$. Each $Part_i$ generates a compatibility tableau (S_i, $Part_i$, $cond_i$), where S_i is the restriction of S_{t_c} and $cond_i$ the restriction of cond over $Part_i$.*

Example 3.1. *Tab. 3 shows the two partitions of the valid partitioning that we obtain in our running example.*

It is easy to prove that the valid partitioning of T_{t_c} is unique and that the following lemma holds.

Lemma 3.1. *Let $Part = \{Part_1, \ldots, Part_k\}$ be the valid partitioning of T_{t_c} and $\mathcal{I}$ be a well-typed database instance. Then,* $CCT(T_{t_c}, \mathcal{I}) = \bigtimes_{Part_i \in Part} CCT(T_{Part_i}, \mathcal{I}_{|Part_i})$.

Using the above lemma, *Ted* first determines the set of concatenated tuples for each partition and then forms the cross product of the tuples of each partition in order to obtain $CCT(T_{t_c}, \mathcal{I})$.

Complexity analysis. The three main phases of *Ted* are the partitioning phase, the computation of concatenated compatible tuples, and the computation of the Why-Not answer. The respective worst case complexities add up to $O(|\mathcal{S}_Q| + \prod_{R \in \mathcal{S}_Q} |\mathcal{I}_{|R}| + \prod_{R \in \mathcal{S}_Q} (|\mathcal{I}_{|R}|) * |\mathcal{S}_Q| * |Q|)$. Assuming that the number of tuples $|\mathcal{I}_{|R}|$ of a relation R is typically much larger than the size of the schema or query (i.e., $|\mathcal{I}_{|R}| >> |\mathcal{S}_Q|$ and $|\mathcal{I}_{|R}| >> |Q|$), this simplifies to $O(\prod_{R \in \mathcal{S}_Q} |\mathcal{I}_{|R}|)$ or $O(N^k)$, where k is the number of relations and N the maximum size of a relation instance.

Implementation and evaluation. We implemented *Ted* in Java 1.6 and ran it over several benchmark queries we defined over three different datasets (the same as in [2]). Due to space constraints and the obvious efficiency issue entailed by *Ted*'s complexity, we only very briefly show one case for each dataset.

Tab. 5 reports *Ted*'s Why-Not answer polynomial and the picky operators identified by NedExplain [2] and Why-Not [3]. These use cases clearly demonstrate that *Ted* returns complete Why-Not answer as opposed to NedExplain and Why-Not, that both in general return subsets of the operators referred to in the polynomial *Ted* returns. This comes at no surprise, as NedExplain and Why-Not base their procedures on a specific query plan in which they trace compatible data until up to the point (query operator) where they disappear. Being query plan independent, Ted produces the complete set of picky operators for all reordered query plans.

Use case	Ted	NedExplain	Why-Not
Crime7	$952op_9op_8 + 8op_9 + 56136op_7op_9op_8 + 792op_7op_9$	op_8, op_9	op_7
Imdb2	$8op_3op_1$	op_3	
Gov2	$17400op_3 + 12op_1 + 19952op_3op_1$	op_1	op_3

Table 5. Ted, NedExplain and Why-Not results

Concentrating on the Why-Not answer polynomials, we see that they bear more information than what previous algorithms return. Indeed, they not only tell us why t_c is missing, but also all the different ways it was pruned from the result. For example, in Crime7, we conclude that the majority of cc-tuples do not satisfy the conditions of the op_7, op_9, and op_8, but we also see 9 cc-tuples that are only pruned by op_9. This information is interesting for subsequent processing (manual or automatic), e.g., we can deduce that the least "invasive" repair of the query touches op_9.

4. Outlook and Future Work

Ted is an algorithm that returns query based-explanations for a Why-Not question over a conjunctive query. Opposed to previous work, it is the first algorithm that is guaranteed to return the same explanations, no matter the considered query plan representation. Another novelty is to represent the Why-Not answer as a polynomial. This polynomial has the benefit of being an elegant formalism that can subsequently be used for further processing, e.g., for ranking the importance of "misbehaving" query operators in the query, for actually computing query rewritings that automatically fix the problem, estimating the minimum number of side-effects of a rewriting, etc. These are interesting problems we plan to address in the future. However, before developing solutions to these interesting problems, we will tackle the problem of efficiency. Besides parallel computations, we may for instance reduce the overall complexity by only selecting a "representative" sample of cc-tuples and compute an approximate result (within certain error bounds).

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0.

[2] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with Nedexplain. In *International Conference on Extending Database Technology (EDBT)*, 2014.

[3] A. Chapman and H. V. Jagadish. Why not? In *International Conference on the Management of Data (SIGMOD)*, 2009.

[4] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007.

[5] Z. He and E. Lo. Answering why-not questions on top-k queries. In *International Conference on Data Engineering (ICDE)*, 2012. .

[6] M. Herschel. Wondering why data are missing from query results? ask conseil why-not. In *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[7] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3 (1), 2010.

[8] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1), 2008.

[9] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4), 1984.

[10] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *International Conference on Data Engineering (ICDE)*, 2013.